PROJECT – REPORT

Introduction :

We were expected to design and implement a processor which supports these instruction set : (AND , ADD , ANDI , ADD , LD , ST , CMP , JUMP , JE , JA , JB , JBE , JAE).

In order to do that we needed to design 16 bits of instruction set architecture (ISA) which declares our way of representing operations.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND	0	0	0	DEST			SRC1			0	0	0	0	SRC2		
ADD	0	0	1	DEST			SRC1			0	0	0	0	SRC2		
ANDI	0	1	0	DEST			SRC1			IMM						
ADDI	0	1	1	DEST			SRC1			IMM						
LD	1	0	0	DEST			ADDRESS									
ST	1	0	1	SRC1			ADDRESS									
CMP	1	1	0	XXXXXX						OP1				OP2		
JUMP	1	1	1	0	0	0	ADDRESS									
JE	1	1	1	0	0	1	ADDRESS									
JA	1	1	1	0	1	0	ADDRESS									
JB	1	1	1	0	1	1	ADDRESS									
JBE	1	1	1	1	0	0	ADDRESS									
JEA	1	1	1	1	0	1	ADDRESS									

This is how we represented our instructions. In order to make opcode easier to read and make our job easier in Assembler , we use 3 bit for the opcode. Then we put registers needed for the operation in order.

For jump instructions , we tought that 10 bits would be enough for the address space and we declared self-defined opcode inside these jump instructions . The reason why we made this decision is that we needed to seperate 3 bits for opcode instead of 4 bits because of load-store addresses.

Assembler :

We used java to create assembly codes from given input set. Assembler's job is to create input file for logism by using our input set.

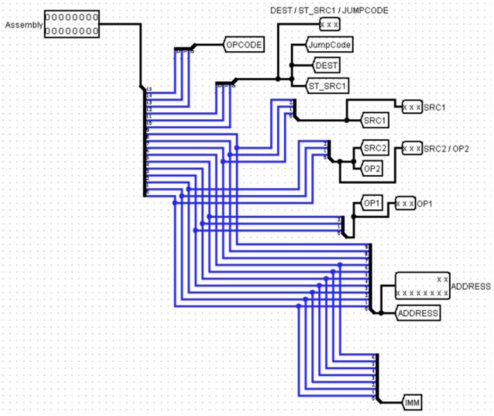
```
private final String[] instructionSet = {"AND", "ADD", "ANDI", "ADDI", "LD", "ST", "CMP" ,"JUMP", "JE", "JA", "JB", "JBE", "JAE"};
private final int immBit = 7 ;
private final int addressBit = 10 ;
```

We used that instruction set. 7 bits for immediate values and 10 bits for addresses are used.
Either address and immediate values are in 2's complement form.

```
switch (parseOpcode[0]) {
    case "AND", "ADD" -> {
        sb.append(registerToBinary(parseReg[0]));    // dest
        sb.append(registerToBinary(parseReg[1]));    // src1
        sb.append("0000");
        sb.append(registerToBinary(parseReg[2]));    // src2
    }
    case "ANDI", "ADDI" -> {
        sb.append(registerToBinary(parseReg[0]));    // dest
        sb.append(registerToBinary(parseReg[1]));    // src1
        sb.append(immToBinart(parseReg[2]));         // imm value
    }
    case "CMP" -> {
        sb.append("0000000");
        sb.append(registerToBinary(parseReg[0]));    // op1
        sb.append(registerToBinary(parseReg[1]));    // op2
    }
    case "LD", "ST" -> {
        sb.append(registerToBinary(parseReg[0]));    // src1
        sb.append(addressToBinary(parseReg[1]));
    }
    case "JUMP" -> {
        sb.append("000");
        sb.append(addressToBinary(parseReg[0]));
    }
    case "JE" -> {
        sb.append("001");
        sb.append(addressToBinary(parseReg[0]));
    }
    case "JA" -> {
        sb.append("010");
        sb.append(addressToBinary(parseReg[0]));
    }
    case "JB" -> {
        sb.append("011");
        sb.append(addressToBinary(parseReg[0]));
    }
    case "JBE" -> {
        sb.append("100");
        sb.append(addressToBinary(parseReg[0]));
    }
    case "JAE" ->{
        sb.append("101");
        sb.append(addressToBinary(parseReg[0]));
    }
}
```

Logisim Design :

CPU :



According to the our ISA design , first 3 bits represents the opcode , next 3 bits represents DEST for ALU and LD operations , SRC1 for ST or JumpCode for JUMP operations. Next 3 bits are used to represent SRC1 for ALU operations. Last 3 bits represents SRC2 for AND and ADD operations , OP2 for ST . [5:3] are used for OP1. Last 10 bits are address and last 7 bits are immediate value.

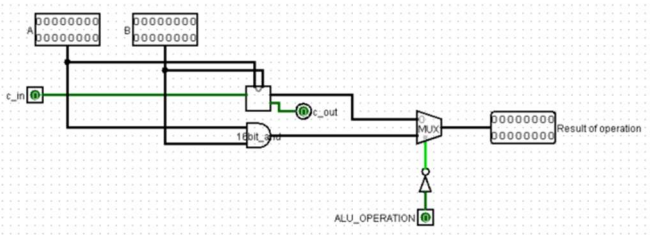
According to the opcode , we determines the operation . If operations are ANDI or ADDI , we produce IMM_SIGNAL for immediate ALU operations. We produce ALU_OP signal to seperate AND , ANDI and ADD,ADDI operations. If operations is AND type , ALU_OP will be zero . Otherwise ALU_OP will be one. Also we procude is_alu signal to determine whether the operation is ALU or not. If operation is CMP , then we produce is_cmp signal as 1 . If operation is LD , we produce MemRead signal as 1 . If operation is ST , we produce MemWrite signal as 1. If operation is JUMP , we produce is_jump signal as 1 . If operation is ALU or LD , we produce register_write_enabled signal as 1.

We give 10-bits address as an output to the main circuit and also we extended immediate value to 16 bits and send it to the main circuit as well.

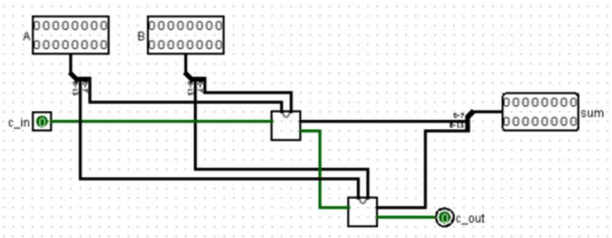
ALU :

We have AND, ADD, ANDI and ADDI operations in ALU. According to The ALU_OPERATION signal , we determine which operation will be processed. If it is zero, that means operation is AND or ANDI. If it is one , that means operation is ADD or ADDI.

ALU



16 Bits Adder

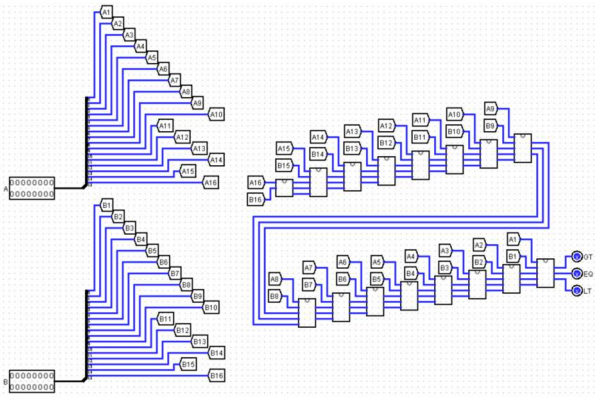


Register File :

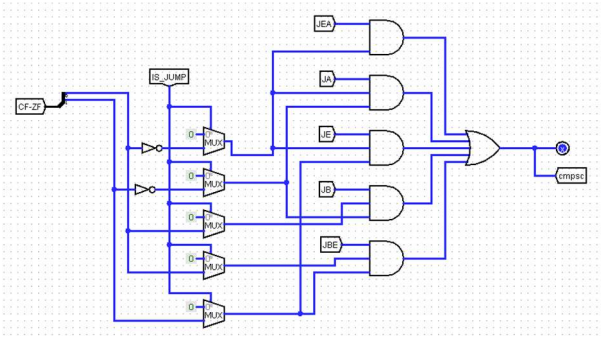
We have 8 registers. Because of that , we used 3 bits to represent each register. Register file includes 8 registers with inputs for read register 1 , read register 2 and write register. Write data value is the input value of register. All register's data bits are 16-bits because of the input data size. "Write Enable" signal controls the register file. Thanks to this signal , signal stores an incoming result in the register indicated by the write register. We have also clear signal to reset all registers in register file. Multiplexers and decoders are used for the selection of registers.

We designed our own registers included clear functionality by using 2x4 multiplexers and D-Flip-Flops .

CMP :



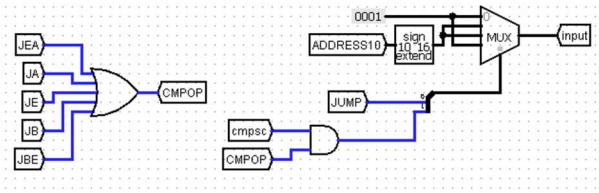
We designed 16-bits comparator by using 1 half bit comparator and 15 full comparator. Half bit comparator takes 2 input and generate gt , lt or eq signal by comparing inputs. Full bit comparator takes 5 input and generate gt , lt or eq signal by using all of these inputs. By using these components , we design our comparator. Our comparator compares input registers and generate flag values according to the result.



According to the CMP result , CF-ZF values are set.

If IS_JUMP signal is 1 , we compare jump instruction's flag values with these CF-ZF values. If they match , then cmpsc signal will be 1 .

According to the this signal , if operation is one of the jump instructions and also cmpsc signal is 1 or just JUMP instruction , then we gives the address value to the counter as an input. Otherwise , our counter is incremented by 1 .

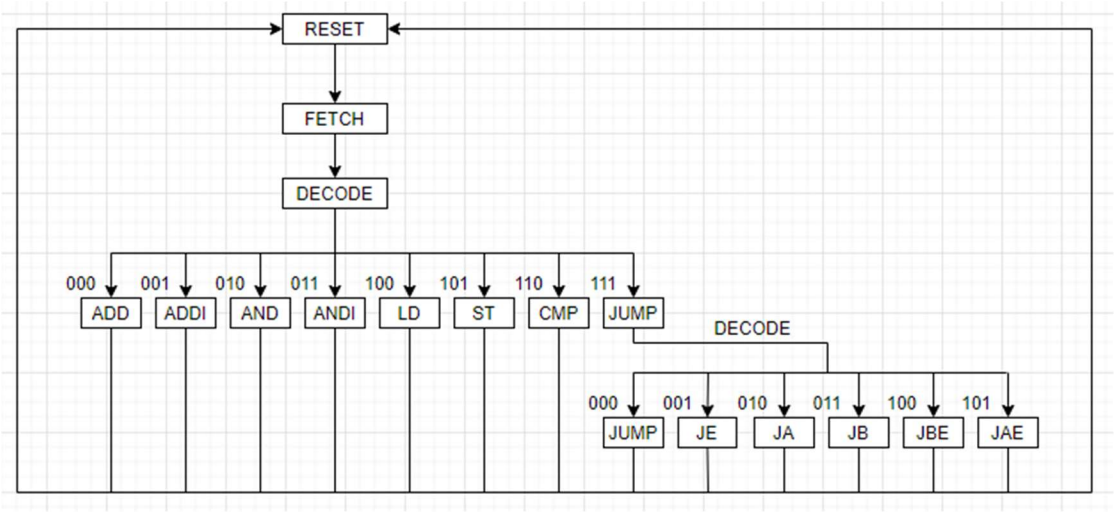


Counter :

According to the input value coming from previous image that we introduced , current address value is incremented by 1 or re-calculated by using input . (output += input) or (output++) .

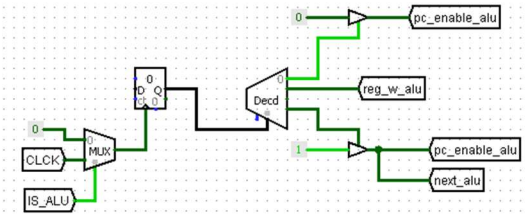
When pc_enable signal which comes from FSM is 1 , this operation is processed.

FSM :



All of our identical instructions have their own execution structure since they are doing different operations. Let's give an example:

We have a structure like the image on the right for most of the instructions. All instructions have its own state number according to the operation . First state is same for all instructions except JUMP instruction and



the purpose of first state is making pc_enable signal zero so that our counter stay same position for processing our current instruction . The last state of FSM is setting pc_enable as 1 .

Data Memory :

We used RAM component as data memory . We have connected 10 bits address input , decides which address to write/read . Reg1_value represents the value read from register , and data_out is the value of the register which pointed by address10. When MEM_WRITE is 1 , it writes the register value into RAM ; if MEM_READ is 1 , it reads from the RAM and store its value into an appropriate register.

Yunus Emre Ertunç – 150117064

Muhammed Enes Aktürk – 150117036

Mahmut Hilmi Arıkmert - 150117024